Please amend the present application as follows:

Specification

The following is a marked-up version of the specification with the language that is underlined ("___") being added and the language that contains strikethrough ("~~——~~") being deleted:

Page 2, line 16 through page 3, line 2.

The information collected by the various instrumentations and program counter sampling (known as a "program counter histogram") normally ~~are~~ is analyzed by a profiling program that generates a user readable call graph, or a visual representation of it, which can be studied by the programmer to learn about the manner in which the application executes. The call graph normally comprises a series of nodes that represent the various application functions, and a series of arcs that connect the nodes and represent the various associations between the application functions. In addition, the call graph can include annotations that provide various information such as the amount of time spent within a particular application function, the number of times a function is invoked, the number of times a function invokes another function, *etc.*

Page 4, line 6 through line 17.

Furthermore, conventional profiling methods do not permit the programmer to limit the collection of information to information concerning only those code portions that are most frequently used because such information is not ~~know~~ known beforehand at compilation time. As is known in the art, programmers typically are not concerned about the execution of functions where the time spent executing them is so minimal as to be nearly irrelevant. However, with

conventional profiling techniques, each function (or other application code portion) is individually instrumented. Another common drawback of traditional profiling schemes is that profiling is restricted to predefined statistical measures as defined by the compiler toolchain or profiling tool used. Therefore, the programmer cannot define other quantities or measures to be done on the application that may be more meaningful to the programmer for a specific application.

Page 27, line 6 through page 28, line 10.

With the collected cycle information, an estimate of time spent in executing each code fragment can be determined as long as the processor speed is known or, in the case a hardware counter is used, the counter frequency. This information can then be included within the call graph (or other output) generated by the profiling program. Although the addition of the instrumentations to the cached code fragments add overhead to application execution, the amount of this overhead is significantly less than that added in conventional profiling methods in that instrumentations are inserted in the form of low-level instructions that have minimal impact on the code. Once the code fragment has been instrumented, it is emitted into the code cache(s) 124, as indicated in block 520, and flow returns to block 508 described above.

The methods described above do not prevent the use of statistical sampling as an alternative method to profiling the code even when the ~~hit~~ hot fragments are executed in the code cache in that the DELI 100 can make a determination as to which original application code is associated with the code in the code cache and produce the correct statistical output for the original code. Inserting instrumentation at run-time also enables user defined quantities to be accumulated for code fragments or functions and/or gives more control to the user over

what is being monitored with a much finer level of control than traditional profiling schemes. For example the user may require filtering of part of the call graph so that the output of the profiling is not cluttered with irrelevant information Most utilities used to display profiling information to programmers allow this as a post pass on the profiling data The advantage of the disclosed approach is that the overhead of event collecting it is removed.

Another example of an alternative quantity that can be profiled, but which often proves difficult to measure, is the amount of instruction level parallelism (ILP). Such a quantity represents the average number of elementary operations (*e.g.*, such as those completed by reduced instruction set computers (RISCs)), executed in parallel per cycle by the processor, to run a given program or application. Such a quantity bears a great importance to analyze program performance for certain kinds of microprocessors such as very long instruction word ~~(VLIWs)~~ (VLIW) or superscalar, and in general any processor that exploits instruction level parallelism.